Containerization on the edge

• 6 minutes to read

This work is supported by Second State and FutureWei based on Open Source projects Wasm-Edge and seL4.

Application containers, such as Docker, are a key driving force behind the growth of Cloud Native applications. However, while the Cloud-Native development paradigm has proven very popular, it is difficult to expand the Cloud-Native infrastructure beyond large data centers since application containers require significant amounts of computing resources. For example, Docker does not support real-time operating systems (RTOS) and only works on POSIX systems.

Furthermore, on edge networks and devices, such as smart factories and smart automobiles, the industry ecosystem and suppliers network dictate that applications must be assembled from multiple independent vendors. For example, in a typical electrical vehicle, there are over 100 suppliers writing software components for different parts of the vehicle. It is crucial for the automobile OEM to provide a secure, high-performance, and real-time runtime environment for suppliers and vendors to integrate their software components. There are already several attempts to support application containers on edge RTOSes.

> VxWorks is a leading commercial RTOS used in mission critical systems such as airplanes and space ships. VxWorks containers is a recent initiative (in 2021) to support OCI-compliant lightweight containers on the VxWorks RTOS.

However, the Docker approach is not a good fit for RTOS on the edge. Fundamentally, Docker is not real-time and assumes the availability of many underlying OS services. A much better runtime approach for RTOS is high-level bytecode VMs. Such VMs could be much lighter and faster than Docker. They provide capability-based secure sandboxes, make very little assumption about the underlying OS services, and at the same time, sup-

port multiple programming languages on the front end. WebAssembly, with its wide industry support and lightweight design, appears to be just the perfect VM runtime for complex edge applications.

> Another interesting aspect of WebAssembly is that WebAssembly programs can often be formally verified for correctness just like seL4 itself. That makes them appropriate for mission critical systems like automotive OSes.

WasmEdge and seL4

The seL4 operating system is a formally verified, highly secure, and real-time micro-kernel operating system. It is now increasingly used in autonomous vehicles and drones where security and real-time performance are critical. The seL4 OS is a micro-kernel and not compliant to POSIX, which makes it especially challenging to run Docker-like containers. WebAssembly, on the other hand, could abstract away much of the operating system and provide developers a unified set of programming languages and SDKs to work with.

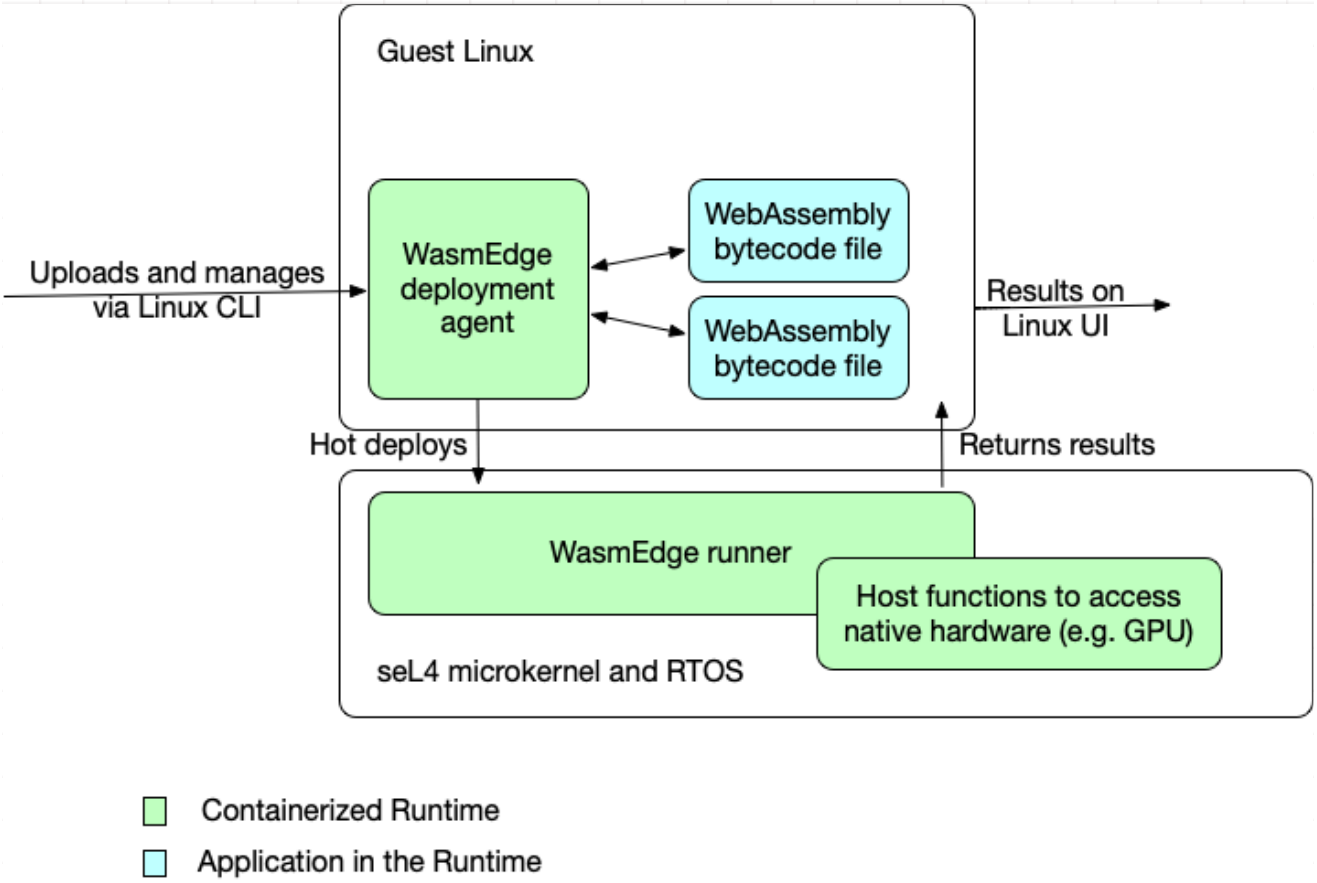
The WasmEdge Runtime is a high-performance and open source WebAssembly runtime hosted by the CNCF. It is used in the cloud native infrastructure as a runtime for microservices, serverless functions, and plugins. Besides standard WebAssembly specs, WasmEdge supports extension APIs that are relevant to cloud native use cases, such as network sockets, Tensorflow-based inference, database storage etc. WasmEdge supports Rust and JavaScript as frontend languages, and can be embedded into Rust, Go, Python, and Node.js host applications as plugins or embedded functions.

Most relevant to the "edge container" use case, WasmEdge is an OCI compliant runtime and can be managed and orchestrated by Docker tools and Kubernetes. In this work, we built a WebAssembly management agent for seL4 and WasmEdge. It allows WebAssembly bytecode applications to be deployed and executed on the seL4 RTOS with ease.

> Officially, seL4 only supports applications written in C/C++. Through WasmEdge Runtime, developers can now use any WebAssembly language to write seL4 RTOS applications, including Rust, Swift, AssemblyScript, and JavaScript. That could be a significant improvement to seL4 developer experience.

Overall design

The seL4 microkernel can function as a hyperviser. It can start a seL4 RTOS and a Linux OS (called guest OS) side by side on the same hardware. The Linux guest OS has a full set of features and tools for file system, networking, user accounts, shell, and CLI, but it is not real-time. The seL4 side is real-time, but headless. Our approach is run the WasmEdge agent in guest Linux. We can upload and store the WasmEdge bytecode file in the guest Linux, and then use the agent to hot deploy and execute the bytecode using a WasmEdge runner installed in seL4. The architecture is as follows.



This agent and runner architecture allows us to combine the guest Linux's ease-of-use with seL4's robustness, security, and real-time performance.

> This design raises an interesting possibility. Maybe we could run a fully fledged Kubernetes pod in the guest OS to manage and orchestrate WasmEdge applications on seL4. That is an area of active research by the team.

Sample WebAssembly apps

WasmEdge can run any WebAssembly bytecode application on seL4. The sample WebAssembly applications in this demo are compiled from C and Rust source code.

- The `nbody-c.wasm` is a program to numerically approximate the N-body problem in C language. It is then compiled into WebAssembly bytecode from C.
- The `hello.wasm` is a Rust program to echo a string to the console.

Patching seL4 for WasmEdge runner

The standard libraries in seL4 do not support WasmEdge runner out of the box. We need to patch those libraries to add, turn on, or update some important features. We build a customized version of seL4 with these patches.

- Patched LLVM compiler
- Patched seL4 system libraries
- Patched guest Linux libraries

A simulator demo

The build script automates the process of building a seL4 distribution with patched libraries, the WasmEdge runner, a guest Linux OS (Ubuntu 20.04), and the WasmEdge agent (called `wasmedge_emit`).

> The build script requires an Ubuntu 20.04 system with developer tools installed. See here for a complete list of apt packages required on the system.

Once the customized seL4 distribution is built, we can run it in a QEMU simulator. We can log into the guest Linux OS's command shell, upload and save WebAssembly bytecode files, and then run wasmedge_emit to deploy and run those WebAssembly files in seL4.

The demo walks you through the entire process. You can watch a video to see it in action!

https://youtu.be/2Qu-Trtkspk

The GitHub Actions log shows the console output from a successful build task, and the artifact contains the build result. You can simply download the build artifact to your own Ubuntu 20.04 machine and start the simulator to run WebAssembly programs on seL4.

What's next

In this article, we demonstrated how to manage and execute WebAssembly applications on seL4 using the simulator. The next step is to run WasmEdge applications on real hardware.

One of the key features of WasmEdge is that it is extensible. It is easy to add host function APIs to WasmEdge from shared native libraries so that WasmEdge WebAssembly bytecode programs can access hardware such as GPIO pins, cameras, USB connectors, I/O boards, and GPUs. Stay tuned for more use case demos for WasmEdge on seL4!

Product WasmEdge RTOS #WebAssembly #RTOS #seL4 #cloud computing #Edge computing

A high-performance, extensible, and hardware optimized WebAssembly Virtual Machine for automotive, cloud, AI, and blockchain applications

second-state @secondstateinc